CBASIC® Compiler (CB80™)

Language Programming Guide

DIGITAL RESEARCH™

CBASIC® Compiler (CB80™)

Language

Programming Guide

COPYRIGHT

DISCLAIMER

Digital Research makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, Digital Research reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Digital Research to notify any person of such revision or changes.

TRADEMARKS

The *CBASIC Compiler (CB80) Language Programming Guide* was prepared using the Digital Research TEX Text Formatter and printed in the United States of America.

First Edition: January 1983

Foreword

The CBASIC® Compiler is a compiler version of the CBASIC programming language. For the software developer interested in maximizing the execution speed of commercial applications programs, the CBASIC Compiler is an excellent choice.

Digital Research designed the CBASIC Compiler for use under single-user, multi-user, and concurrent operating systems based on both 8-bit and 16-bit microprocessors.

- 8-bit CBASIC Compiler, CB80™, runs under the CP/M® versions 2 and 3, MP/M™, and CP/NET® operating systems based on the Intel® 8080, 8085, or Zilog Z80® microprocessor.
- 16-bit CBASIC Compiler, CB86™, runs under the CP/M-86®, MP/M-86™, and Concurrent CP/M-86™ operating systems based on the Intel 8086, 8088 family of microprocessors.

The *CBASIC Compiler Language Programming Guide* provides a short demonstration program to help you get your CBASIC Compiler system up and running. The manual is divided into five sections.

- Section 1 is an introduction and demonstration program.
- Section 2 describes the compiler, CB80.
- Section 3 describes the link editor, LK80™.
- Section 4 describes the indexed library and the library manager utility program.
- Section 5 explains the machine-level environment of the CBASIC Compiler.

Use this Programming Guide in conjunction with the *CBASIC Compiler Language Reference Manual.* Together, the manuals provide all the information you need to use the CBASIC Compiler to its full potential.

Digital Research is very interested in your comments on programs and documentation. Please use the Software Performance Reports and the Reader Comment card to help us provide you with the best microcomputer software and documentation.

Table of Contents

Table of Contents (continued)

Appendixes

Table of Contents (continued)

List of Tables

List of Figures

Section 1
Getting Started with the CBASIC Compiler

A compiler is a computer program that translates high-level programming language instructions into machine readable code. The compiler takes as input a user-written source program and produces as output a machine-level object program. Some compilers translate a user-written source program into a program that a computer can execute directly. The CBASIC Compiler system, however, uses a link editor and a library in addition to the compiler. Together, the three components translate your CBASIC source code file into a directly executable program using your microcomputer's memory space as efficiently as possible. The system enables you to modularize programs for quick and easy maintenance. The result is a programming system that rivals the performance of systems based on much larger machines.

The primary advantage that compilers provide over other methods of translation is speed. Compiled applications programs execute faster than interpreted programs because the compiler creates a program that the computer can execute directly.

1.1 Components

The three components that make up the CBASIC Compiler system are listed in the directory of your CBASIC product disk along with three compiler overlay files and the library manager utility:

- The compiler, CB80, translates CBASIC source code into relocatable machine code modules. Source programs default to a .BAS filetype unless specified otherwise. CB80 generates .REL files.
- The link editor, LK80, combines the relocatable object modules that the compiler creates and relocatable routines from the library into a directly executable program with optional overlays. LK80 generates .COM files.
- The library provides relocatable routines that allocate and release memory, determine available memory space, and perform arithmetic operations and input/output processing.

1.2 A Demonstration

The following demonstration program can help you learn how to compile, link, and run your first CBASIC program. The instructions are for the CBASIC Compiler on a CP/M-based system with two floppy-disk drives. You should already be familiar with CP/M and a text editor.

Make a back-up copy of your master CBASIC Compiler product disk. Place your operating system disk in drive A and a copy of your CBASIC Compiler disk in drive B.

1. Write the source program.

   Using your text editor, create a file named TEST.BAS on your CBASIC Compiler disk in drive B. Enter the following program into TEST.BAS exactly as it appears below:

```
PRINT
FOR I% = 1 TO 10
    PRINT I%; "TESTING THE CBASIC COMPILER!"
NEXT I%
PRINT
PRINT "FINISHED"
END
```

2. Compile the program.

   To start CB80, enter the following command. Be sure drive B is the default drive.

```
B>CB80 TEST
```

   CB80 assumes a filetype of .BAS for the file you specify in the compiler command line unless otherwise specified. A sign-on message, a listing of your source program, and several diagnostic messages display on your terminal.

```
-----------------------------------------
CBASIC Compiler CB80              Version 1.x
Ser. No. 000-0000        Copyright (c) 1982
Digital Research, Inc.  All rights reserved
-----------------------------------------
end of pass 1
end of pass 2
    1:  003eh PRINT
    2:  0041h FOR I% = 1 TO 10
    3:  004ah   PRINT I%; "TESTING THE CBASIC COMPILER!"
    4:  0056h NEXT I%
    5:  0064h PRINT
    6:  0067h PRINT "FINISHED"
    7:  0070h END
end of compilation
no errors detected
code area size:    112        0070h
data area size:    2          0002h
common area size:  0          0000h
symbol table space remaining: 31890
```

Section 2.1 describes the various parts of the listing. The message no errors detected indicates a successful compilation. CB80 creates a relocatable file for the TEST.BAS program. The directory for disk B should have the new file TEST.REL.

3. Link the program.

   To start LK80, enter the following command. Be sure drive B is the default drive.

   ```
   B>LK80 TEST
   ```

   LK80 assumes a filetype of .REL for the file you specify in the linker command line. A sign-on message and several diagnostic messages display on your terminal.

```
---------------------------------------
LK80                          Version 1.x
Ser. No. 000-0000      Copyright (c) 1982
Digital Research, Inc.  All rights reserved
---------------------------------------
code size:       1200 (0100-12FF)
common size:     0000
data size:       0166 (1300-1465)
symbol table space remaining:   124F
```

If you get no error messages, the program has been linked successfully. LK80 creates a directly executable program. The directory for disk B should have the new file TEST.COM.

4. Run the program.

   To run the TEST.COM program, enter the following command. Be sure drive B is the default drive.

```
B>TEST
```

The following output should appear on your terminal:

```
1 TESTING THE CBASIC COMPILER!
2 TESTING THE CBASIC COMPILER!
3 TESTING THE CBASIC COMPILER!
4 TESTING THE CBASIC COMPILER!
5 TESTING THE CBASIC COMPILER!
6 TESTING THE CBASIC COMPILER!
7 TESTING THE CBASIC COMPILER!
8 TESTING THE CBASIC COMPILER!
9 TESTING THE CBASIC COMPILER!
10 TESTING THE CBASIC COMPILER!

FINISHED
```

End of Section 1

Section 2
The Compiler, CB80

CB80 consists of an executable file with filetype .COM and three overlay files. Your CBASIC Compiler product disk should contain the following four files:

- CB80.COM
- CB80.OV1
- CB80.OV2
- CB80.OV3

When compiling a CBASIC program, all four files must be on the logged-in drive. The source program file can be on any logical drive.

2.1 Compiling Programs

CB80 takes a CBASIC source program as input and generates a relocatable object file. During compilation, CB80 creates the following temporary work files with a .TMP filetype:

```
PA.TMP
QCODE.TMP
DATA.TMP
```

Unless compilation is unsuccessful, you never see these temporary files listed in a directory. CB80 erases the files automatically when compilation is finished. CB80 also erases the temporary files if they are on disk before you start the compiler.

The size of the .TMP files varies according to the size of the source program. The amount of temporary space required is approximately equal to the amount of space the source program occupies. If you do not have enough work space on disk for the compiler, you can break up large programs into modules and compile each module separately.

The following is an example of a CB80 listing:

```
-----------------------------------------
CBASIC Compiler CB80              Version 1.x
Ser. No.                    Copyright (c) 1982
Digital Research, Inc.  All rights reserved
-----------------------------------------
end of pass 1
end of pass 2
    1:  003eh PRINT
    2:  0041h FOR I% = 1 TO 10
    3:  004ah   PRINT I%; "TESTING THE CBASIC COMPILER!"
    4:  0056h NEXT I%
    5:  0064h PRINT
    6:  0067h PRINT "FINISHED"
    7:  0070h END
end of compilation
no errors detected
code area size:     112         0070h
data area size:     2           0002h
common area size:   0           0000h
symbol table space remaining: 31890
```

Certain phases of the compilation process are combined into a module called a pass. CB80 is a three-pass compiler. Following the sign-on message, CB80 indicates the completion of the first two passes with a message. The program listing includes the line numbers, relative addresses for the code that each line generates, and the actual source code lines. In the preceding listing, 1: is an example of a line number. 003eh is a relative address for the relocatable code that the first PRINT statement generates.

CB80 prints the total number of compilation errors detected in the program following the message end of compilation. The message no errors detected, however, indicates a successful compilation. The last four messages indicate the amount of space CB80 allocates for certain segments of data. Refer to Section 5 for an explanation of memory allocation. If CB80 detects errors, the relative addresses and the memory allocation messages do not print.

To complete the compilation process, CB80 generates a relocatable object file. The relocatable file has the same filename as the source program and has a .REL filetype. The .REL file requires approximately the same amount of space as the source program. If the source program contains errors that prevent a successful compilation, CB80 does not generate the .REL file.

2.1.1 CB80 Command Lines

The command line starts CB80, specifies the file to compile, and passes special information in the form of compiler directives. The following command line compiles the source program in a file named TEST. CB80 assumes a filetype of .BAS unless otherwise specified.

```
CB80 TEST
```

Enter a complete file specification to override the .BAS filetype. The following command line compiles the source program in a file named TEST.PR1. Remember, source files cannot have a .REL filetype.

```
CB80 TEST.PR1
```

Source files can be on any logical disk drive. The following command line compiles the source program TEST.PR1 from drive D:

```
CB80 D:TEST.PR1
```

If you type an incorrect command line or neglect to enter a filename, CB80 indicates the error with the message invalid command line.

2.1.2 Compiler Errors

CB80 reports three different types of compiler errors. The first type, file system and memory space errors, includes mistakes such as invalid command lines, read errors, and out of memory conditions. CB80 indicates file system and memory space errors with literal messages such as disk full and symbol table overflow. Refer to Appendix B, Table B-1, for a complete listing of file system and memory space error messages.

The second type, compilation errors, includes misuses of the CBASIC language such as invalid characters, improper data type specifications, and missing delimiters. CB80 inserts an integer value in the compiler listing of the source program to indicate the occurrence of a compilation error. The integer corresponds to an error description listed in Appendix B, Table B-2.

The third type, fatal compiler errors, should never occur during your experience with the CBASIC Compiler. CB80 indicates a fatal compiler error with the following message. The XXX stands for a three-digit integer value.

```
FATAL COMPILER ERROR XXX
NEAR SOURCE LINE XXXX
```

If this error message occurs during compilation of your CBASIC program, contact the Digital Research Technical Support Center. Please report the three-digit integer and the circumstances under which the error occurs.

2.2 Compiler Directives

Compiler directives are special instructions to CB80. The CBASIC Compiler supports two different ways to specify compiler directives: source code compiler directives and command-line toggles.

2.2.1 Source Code Compiler Directives

Source code compiler directives are special keywords that do not translate into executable code. All source code compiler directives begin with a percent sign. You cannot place blanks between the percent sign and the rest of the keyword. Only blanks and tab characters can precede a directive. Source code compiler directives cannot appear on the same line with CBASIC statements or functions. CB80 ignores all characters on the same line that are not part of the directive. A source code compiler directive cannot span more than one line with a continuation character. You cannot label source code compiler directives.

The CBASIC Compiler supports the following six source code compiler directives:

- %NOLIST
- %LIST
- %EJECT
- %PAGE
- %INCLUDE
- %DEBUG

Normally, CB80 generates a listing of the source program during compilation. The %NOLIST directive tells CB80 not to list anything that follows the %NOLIST in the program. The %LIST directive tells CB80 to resume the listing. Use %LIST and %NOLIST any number of times in a program. Toggle B described in Section 2.2.2 suppresses all listings regardless of any directives in the source code.

The %EJECT directive tells CB80 to continue the program listing at the top of the next page of paper. %EJECT works only when you direct the listing to a printer. CB80 ignores %EJECT if the %NOLIST directive is in effect, or if you direct the listing to the console or a disk file.

The %PAGE directive sets the page length for a listing directed to a printer. The page length you specify must be an unsigned integer placed after the %PAGE keyword, as shown in the following example:

```
%PAGE 40
```

The %INCLUDE directive tells CB80 to include the code from a specified source file along with the original compiling program. The included source file is incorporated into the original program immediately following the %INCLUDE. Specify the filename, the filetype, and the drive that holds the file. CB80 assumes the default drive and a .BAS filetype if not specified otherwise. The following examples show three variations of %INCLUDE:

```
%INCLUDE CONDEF

%INCLUDE CONDEF.INC

%INCLUDE D:CONDEF.INC
```

You can nest included files six deep. The maximum nesting depth depends on your particular implementation of the CBASIC Compiler. Refer to Appendix A for current implementation dependent values.

The %DEBUG directive works with three command line toggles: the I, N, and V toggles. You can switch these three toggles on or off from within the program source code. To turn a toggle on, place the toggle letter after the %DEBUG keyword. To turn a toggle off, place the toggle letter preceded by a minus sign after the %DEBUG keyword. The following examples show variations of the %DEBUG directive:

```
%DEBUG I

%DEBUG -I

%DEBUG INV

%DEBUG -I-N-V
```

2.2.2 CB80 Command Line Toggles

Command line toggles are single-letter compiler directives that you specify in the CB80 command line instead of in the source program. Once a toggle is set, it normally remains set through the entire compilation process. The %DEBUG directive can change the I, N, and V toggles during compilation. Place letters within brackets following the file specification in a CB80 command line. Letters can be lower- or upper-case. If you enter conflicting toggles in a command line, the last one read from left to right takes effect. Certain toggles require an additional parameter enclosed in parentheses. The following examples show several ways to specify command line toggles:

```
CB80 TEST [B]

CB80 TEST.BAS [B, P, S]

CB80 FILE.DAT [BPW(72)]

CB80 CALCS.PRG [N] [O] [P]

CB80 DATA.OVL [ Pon ]
```


CB80 supports the fifteen command line toggles listed in the following table:

Table 2-1. CB80 Toggles

Toggle	*Instruction*
B	Suppress listing of the source file.
C	Change the default %INCLUDE file disk.
F	Send the source listing to a disk file on the same drive as the source file.
I	Interlist the generated code with the source file.
L	Set the page length for printed listings.
N	Generate code for line numbers.
O	Suppress the generation of the object .REL file.
P	List the source file on the printer.
R	Change the disk that the .REL file is written to.
S	Include symbol name information in the .REL file.
T	List the symbol table following the source listing.
U	Generate error messages for undeclared variables.
V	Put source code line numbers into the .SYM file.
W	Set the page width for printed listings.
X	Change the disk used for the work files.

The B toggle tells CB80 not to list the source program on the console screen. However, compiler errors and statistical data concerning size of code and data areas display on the screen. The B toggle overrides other toggles that control compiler output.

The C toggle specifies the default drive for include files. Enclose the new drive specification in parentheses following the C. If a drive has been specified in the %INCLUDE directive, the C toggle has no effect. The C toggle allows program development to be independent of your hardware configuration.

The F toggle tells CB80 to send the source listing to a disk file that is on the same drive as the source file. The new file has the same filename as the source file and has a .LST filetype.

The I toggle interlists compiler-generated code with the original source statements. Compiler-generated code uses standard 8080 mnemonics.

The L toggle changes the page length for a listing directed to a printer. Enclose the new length in parentheses following the L. The length must be an unsigned integer, as in the following example:

```
CB80 TEST [L(50)]
```

The N toggle generates code that saves the current line number for each physical line in a source program. The code enables the ERRL function to return the line number when an execution error occurs.

The O toggle tells CB80 not to generate the relocatable object file. If a compiler error occurs, CB80 does not generate the .REL file.

The P toggle prints the program listing on the printer. CB80 sends a form-feed before printing the first page. CB80 prints the page number and the source filename at the top of each page.

The R toggle specifies which drive to place the .REL file on.

The S toggle places all information on program variables and line labels into the .REL file. The link editor uses the information to generate a .SYM file. You can use the .SYM file with the Digital Research Symbolic Instruction Debugger, SID™.

The T toggle lists the symbol table immediately following the source program listing.

The U toggle generates an error message if a variable name does not appear in an INTEGER, REAL, or STRING declaration. Use the U toggle to locate misspelled identifiers.

The V toggle places the source code line numbers into the .SYM file.

The W toggle changes the page width for a listing directed to the printer. Initially, the width is set to 80 columns. Enclose the new width in parentheses following the W. The width must be an unsigned integer.

```
CB80 TEST [W(70)]
```

The X toggle specifies a drive for the temporary work files. Normally, CB80 places the work files on the same drive as the source file. Enclose the new drive specification in parentheses following the X. The drive is specified by a single lower- or upper-case letter.

```
CB80 TEST [X(D)]
```

CB80 evaluates toggles from left to right. This means a subsequent directive can override any earlier one. In the following example, CB80 sends the listing to the printer.

```
CB80 TEST [BP]
```

In the following example, CB80 suppresses the listing.

```
CB80 TEST [PB]
```

End of Section 2

Section 3
The Link Editor, LK80

LK80 is a linkage editor designed specifically for use with CB80. LK80 combines relocatable object modules that CB80 generates with relocatable modules from the indexed library, CB80.IRL, into an executable program with optional overlay files. Your CBASIC Compiler product disk should contain the following two files:

```
LK80.COM
CB80.IRL
```

When linking a CBASIC program, both files must be on the default drive.

3.1 Linking Modules

LK80 converts a .REL file into an executable .COM file. During linking, LK80 automatically searches the default disk for the indexed library file. LK80 includes any library routines that the compiled program requires in the executable program.

Following a sign-on message, LK80 prints four messages that indicate the amount of space LK80 allocates for the program. Refer to Section 5 for an explanation of memory allocation. The following example shows the console display during linking:

```
-----------------------------------------
LK80                              Version 1.x
Ser. No. 000-0000      Copyright (c) 1982
Digital Research, Inc.  All rights reserved
-----------------------------------------
code size:      1200 (0100-12FF)
common size:    0000
data size:      0166 (1300-1465)
symbol table space remaining:   124F
```

LK80 determines and displays the four values that follow the sign-on message as hexadecimal numbers. The values in parentheses are the memory location assigned to each area.

To complete the linking process, LK80 generates the executable program and a symbol location file with filetype .SYM. The executable program has the same filename as the first .REL file listed in the LK80 command line and has a filetype of .COM. You can specify a different filename for the executable program in the command line. You can use the .SYM file with the Digital Research symbolic debugging program, SID.

3.1.1 LK80 Command Lines

The command line starts LK80 and specifies the relocatable files to link. The following command line links the modules in a file named TEST.REL with the runtime subroutine library and generates an executable program file named TEST.COM. LK80 assumes a filetype of .REL if not specified otherwise.

```
LK80 TEST
```

You can rename a file in the LK80 command line using an equal sign. The following command line links the modules in the file named TEST.REL but generates an executable file named TESTPGM.COM.

```
LK80 TESTPGM=TEST
```

You can specify which drive holds the .REL file to link, and you can specify a drive for LK80 to write the executable file to. The following command line produces the same executable file as in the previous example, but LK80 links the TEST.REL file from drive D and writes the TESTPGM.COM file to disk A.

```
LK80 A:TESTPGM=D:TEST
```

LK80 can link any program that occupies less than 64K bytes of memory unless the length of symbols exhausts the space reserved for the symbol table. You can link several relocatable files into one executable program. However, when combining several files, only one file can contain executable statements. All other files must contain only multiple-line functions. In the following command line, TEST is the executable program, and ONE, TWO, and THREE contain multiple-line functions. LK80 links all four relocatable files into one executable program named TEST.COM.

```
LK80 TEST, ONE, TWO, THREE
```


You can specify a filetype other than .REL for files in the command line if the files are relocatable object files. LK80 aborts the linking process if any of the files are not relocatable object files. In the following command line, TEST.A is the executable program, and ONE.B, TWO.C, and THREE.D contain multiple-line functions. LK80 assumes all four files are relocatable object files and links them into one executable program named TESTPGM.COM.

```
LK80 TESTPGM=TEST.A,ONE.B,TWO.C,THREE.D
```

If you generate your own subroutine library using LIB86, you must specify the library in the LK80 command line. The following example links the library LIB1.IRL into TESTPGM, along with ONE.REL and TWO.REL.

```
LK80 TESTPGM=ONE,TWO,LIB1.IRL
```

Note that you must specify the .IRL filetype for libraries.

LK80 can link up to 60 relocatable files at one time. However, the total length of the command line cannot exceed 128 characters. In cases where a command line exceeds 128 characters, you can shorten filenames, or you can place the command line in a disk file. There is no limit on the length of a command line if LK80 reads it from a disk file.

Create the command line file using any text editor. Do not enter the characters LK80 in the disk file. List each relocatable object file as you would in an ordinary command line. You can place tab characters, carriage returns, and line-feeds anywhere in a command line file. Use the backslash to document large command line files. LK80 ignores all characters that follow a backslash on the same line. Then specify the disk file in an LK80 command line as shown in the following example:

```
LK80 $ CMDLINE.LIN
```

The preceding example tells LK80 to read the rest of the command line from a disk file named CMDLINE.LIN. The dollar sign must follow the LK80. At least one space must separate the dollar sign from the file specification. The command line file can have any filename and filetype.

3.1.2 LK80 Errors

LK80 reports two different types of errors. The first type includes mistakes such as improper command lines and out of memory conditions. LK80 indicates these errors with a literal message. Refer to Appendix C, Table C-1, for a complete listing of LK80 error messages and descriptions.

The second type, LK80 failures, should never occur during your experience with the CBASIC Compiler. LK80 indicates a failure with the following message. The N stands for an integer value.

```
LK80 FAILURE N
```

If the above error message occurs during linking of your CBASIC program, contact the Digital Research Technical Support Center. Please report the number and the circumstances under which the error occurs.

3.2 LK80 Toggles

LK80 toggles are single-letter directives that you specify in the LK80 command line. Once a toggle is set, it remains set through the entire linking process. Place the letters within brackets following the relocatable file specifications in the LK80 command line. Letters can be lower- or upper-case. The following examples show several ways to specify LK80 toggles:

```
LK80 TEST[Q]

LK80 TESTPGM=TEST,ONE,TWO,THREE[QL]

LK80 TEST [M,OB,L]

LK80 TEST [MOBL]

LK80 TEST [M] [OB] [L]
```

LK80 supports five toggles as listed in the following table.

Table 3-1. LK80 Toggles

Toggle	*Instruction*
L	Redirect console output from LK80 to the printer.
M	List all module names followed by an absolute starting address for each.
O	Write output files to a drive other than the default drive.
Q	Place all symbols beginning with a question mark into the .SYM file.
S	Save linking information for the overlays in a file for future short-links.

The L toggle directs all console messages output during linking to the printer.

The M toggle lists all module names and corresponding absolute addresses on the console. LK80 lists library modules first, followed by overlay modules. The M toggle displays a load map that you can use with the addresses provided in the CB80 listing to aid in debugging.

The O toggle directs all output files to a disk drive other than the default drive. Place the new drive specification immediately after the O. For example, the toggle OC writes all output files to drive C.

The Q toggle tells LK80 to place all symbols beginning with a question mark into the .SYM file. The toggle adds about 100 symbols to the .SYM file. If you do not specify the Q toggle, LK80 places only program defined symbols into the .SYM file.

The S toggle provides a way to short-link an overlay file independent of all other overlays in a program. Short-linking enables you to modify an overlay file and relink it into the original program without relinking the entire program. The S toggle saves linking information for the overlays in a disk file. LK80 saves the information in a file with the same name as the root program, but with a .LNK filetype. The following command line links the relocatable files TEST.REL, ONE.REL, and TWO.REL into an executable program TEST.COM and an overlay MSGS.OVL. The S toggle saves information for the overlay.

```
LK80 TEST,ONE,TWO,(MSGS)[S]
```

If you find an error in MSGS.OVL during execution of the program, you can correct the error in MSGS.BAS. Then recompile MSGS.BAS to generate MSGS.REL. Finally, you can short-link the new MSGS.REL into the TEST.COM program using the information saved in TEST.LNK. You do not have to repeat the whole link specified in the original command line. The following short-link command line relinks the new MSGS.REL overlay into TEST.COM. Note that you must not enclose MSGS in parentheses in a short-link command line.

```
LK80 TEST.LNK, MSGS
```

The following rules apply to short-linking with the S toggle:

- The new overlay must not require code, common, or data segment sizes larger than the segment sizes allocated in the original link.
- The new overlay must not reference library modules that are not included in the root. LK80 informs you if this occurs.
- You must place the saved .LNK file first in the LK80 short-link command line. LK80 assumes that all filenames after the .LNK filename constitute one overlay.
- LK80 can short-link only one overlay at a time.
- LK80 does not generate a symbol file during a short-link.

3.3 Producing Overlays

LK80 can produce overlay files that a CBASIC CHAIN statement can load into memory and execute. Overlay files have a .OVL filetype. LK80 overlay files preserve all variables declared in COMMON including variables stored dynamically, such as arrays and strings.

To generate an overlay, enclose the filename of the relocatable object file in parentheses within the LK80 command line. The following command line creates an executable file named TEST.COM and one overlay named ONE.OVL.

```
LK80 TEST(ONE)
```

The TEST.COM file is the root program. A CHAIN statement in the TEST.COM program loads the overlay ONE.OVL. When the root program chains to an overlay, the overlay actually replaces and overwrites the root in memory. This also occurs when an overlay chains to another overlay or back to the root.

The root program contains all library routines for the entire program. This reduces the size of an overlay file and the time required to load an overlay into memory.

LK80 can create up to 60 overlays at one time. However, the total number of relocatable modules in one link cannot exceed 60. The following command line generates an executable program named TESTPGM.COM and two overlays named ONE.OVL and TWO.OVL:

```
LK80 TESTPGM=TEST(ONE)(TWO)
```

You can combine several relocatable modules into one overlay. The following command line generates an executable program named TEST.COM and three overlays named A.OVL, C.OVL, and F.OVL:

```
LK80 TEST(A,B) (C,D,E) (F)
```

You can specify names for the overlay files in the command line. The following command line generates the TESTPGM.COM program and two overlays named FIRST.OVL and SECOND.OVL:

```
LK80 TESTPGM=TEST (FIRST=A) (SECOND=B,C)
```

3.4 Linking Assembly Language Routines

LK80 can link assembly language routines with relocatable modules that CB80 creates. You can use the Digital Research Relocating Macro Assembler, RMAC™, to convert your assembly language programs into relocatable modules that LK80 can link.

Assembly language routines linked into a CBASIC program must not contain initialized data. You can place all data that requires an initial value in the code segment. Refer to section 5.3 for information on parameter passing and returning values.

Note that using assembly language routines makes a program machine-dependent.

End of Section 3

Section 4
The Library

A library file consists of one or more relocatable modules. However, to be useful in the linking process, a library file must contain an index. The CBASIC Compiler provides an indexed library file for use with LK80 and a library manager utility program to create your own library files. Your CBASIC Compiler product disk should contain the following two files:

```
CB80.IRL
LIB.COM
```

4.1 CB80.IRL

The file CB80.IRL is an indexed library file that contains modules to allocate and release memory, determine available memory space, and perform arithmetic operations and input/output processing. All indexed library files have a .IRL filetype. An index precedes the group of modules and contains all the public symbols that are in each module. The index enables LK80 to determine which routines in CB80.IRL are required to create the executable program.

LK80 first reads the .REL files you specify in the command line. LK80 then searches the index of CB80.IRL for any symbols that remain unresolved. LK80 links only those modules from CB80.IRL that contain definitions of the unresolved symbols.

For example, if a module in one of your programs requires the square root subroutine, LK80 searches the index of the CB80.IRL file for the symbol ?RSQR. Assuming that this symbol is not defined anywhere in your program, LK80 links the module from CB80.IRL that contains the definition of ?RSQR. LK80 links any module from the indexed library that contains a required symbol definition.

4.1.1 Dynamic Storage Allocation Routines

The CBASIC Compiler indexed library file provides four routines for use in assembly modules that enable you to allocate and release memory, and to determine the amount of space that is available for allocation.

- The ?GETS routine allocates space. The routine requires that the number of bytes of memory to allocate pass in registers H and L. The maximum number of bytes the routine can allocate is 32,762. ?GETS returns a pointer to a contiguous block of memory in registers H and L. There is no restriction on what the allocated memory space can contain, if the adjacent space at either end of the allocated area is not modified.
- The ?RELS routine releases previously allocated memory. The routine requires that the address of the space to release passes in registers H and L. ?RELS does not return a value.
- The ?MFE routine returns the size of the largest contiguous area available for allocation using the ?GETS routine. The value returned is an integer placed in in registers H and L.
- The ?IFRE routine returns the total amount of unallocated dynamic memory. The returned value is an integer placed in registers H and L. A negative value indicates a number larger than 32,767.

4.1.2 Arithmetic Routines

The CBASIC Compiler indexed library file provides routines for signed integer multiplication and division for use in assembly modules.

- The ?MIDH routine multiplies the signed integer in registers D and E by the signed integer in registers H and L. The routine returns the result in registers H and L.
- The ?DIDH routine divides the signed integer in registers D and E by the signed integer in registers H and L. The routine returns the result in registers H and L.

4.2 The Library Manager Utility, LIB

The LIB.COM file is a versatile librarian program used to develop library files for use with LK80. LIB.COM can perform the following four tasks:

- concatenate a group of .REL files into a library
- create an indexed library, .IRL file
- select modules from a library
- print module names and public symbols from a library

The LIB.COM program supports three command line switches. A switch is a single lower- or upper-case letter that you specify in brackets following the first filename in the LIB command line.

- The I switch creates an indexed library, .IRL file.
- The M switch prints a listing of module names from a specified file.
- The P switch prints a listing of both module names and public symbols from a specified file.

4.2.1 LIB Manager Command Lines

You can concatenate a group of .REL files to unite modules that must execute in combination. Two separate modules might contain public functions that a third module needs for execution. Combining the three modules into one .REL file simplifies the LK80 command line that refers to them. The following librarian command line creates a file named TEST.REL by concatenating the files TEST1.REL, TEST2.REL, and TEST3.REL:

```
LIB TEST=TEST1,TEST2,TEST3
```

The librarian program only concatenates the three original files. The librarian does not modify the files in any other way.

Using the I switch, you can create an indexed library file for a group of modules that support the same types of applications, but are not interdependent for execution. Certain programs might require only one or two modules from a group. Generate an indexed library with the group of modules using LIB.COM. The following command line generates an indexed library named TEST.IRL from the three modules TEST1, TEST2, and TEST3:

```
LIB TEST[I]=TEST1,TEST2,TEST3
```

Specify the library name in the LK80 command line whenever the application program requires certain modules. LK80 searches the index of the library for the required routines and links the corresponding modules into the program. This procedure helps keep the executable program file as small as possible. You can specify up to ten indexed library files in an LK80 command line. LK80 processes library files in order of occurrence.

When you start the LIB.COM program, a sign-on message and some diagnostic messages display on the console. Using the M toggle, you can have LIB.COM generate a list of all module names in the specified files following the diagnostic messages. The following command line generates a list of all modules for the TEST.REL file:

```
LIB TEST[M]
```

Using the P toggle, you can have LIB.COM generate a list of all module names and public symbols in the specified files following the diagnostic messages. The following command line generates a list of all modules and public symbols for the TEST.REL file:

```
LIB TEST[P]
```

End of Section 4

Section 5
Machine-level Environment

To understand the CBASIC Compiler machine-level environment, you should have a working knowledge of CP/M and a familiarity with elementary computer architecture.

5.1 Memory Allocation

The operating system loads an executable CBASIC program into the CP/M Transient Program Area (TPA). Before a CBASIC program loads, the CP/M Console Command Processor (CCP) resides at the top of the TPA. The CBASIC program overwrites all of the CCP after it begins execution.

The following diagram shows memory allocation during execution of a compiled CBASIC program. The area extending from the base of memory at 0H to the base of the TPA at 100H is reserved for CP/M and remains fixed. The area extending from the top of the TPA to the top of memory at FFFFH is reserved for CP/M and remains fixed. When CP/M loads a CBASIC program, the memory available in the TPA is partitioned into six areas of varying size. The diagram shows the relative positions of the different areas in memory, but does not accurately represent relative sizes.

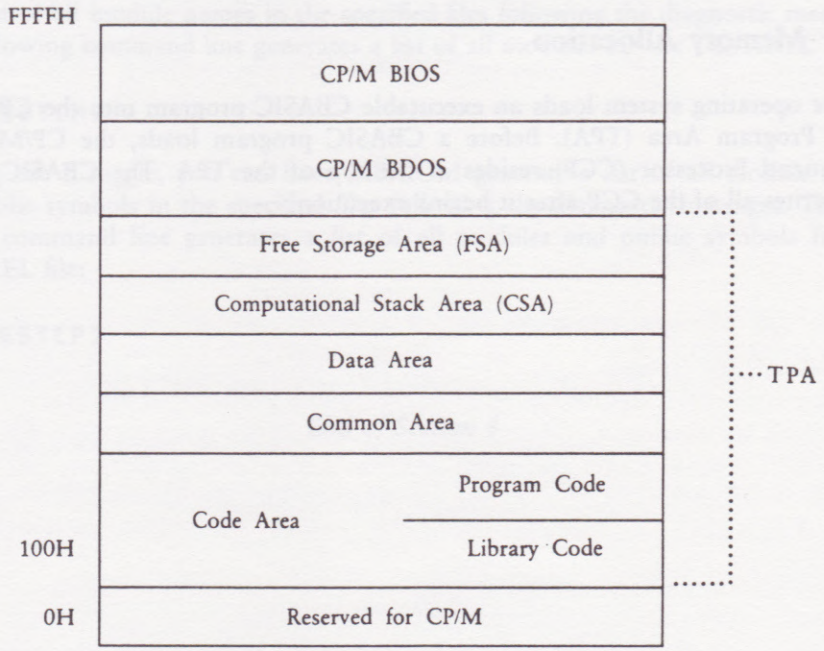

Figure 5-1. CP/M Memory Allocation

- The Code Area contains the actual computer instructions used during execution. The Code Area consists of two partitions: the Program Code and the Library Code. The Program Code section contains the root program. When you chain to an overlay, the overlay file overwrites the root program in this area. Likewise, when you chain back to the root programs, the root program overwrites the overlay file. The Library Code section contains the various routines from CB80.IRL and other indexed library modules that the program requires for execution.
- The Common Area contains all variables passed through COMMON statements to chained programs. The Common Area reserves eight bytes of storage space for each variable, regardless of data type. For array and string variables, the actual value is stored in the Free Storage Area. The value stored in the Common Area is an address in the FSA.
- The Data Area contains all variables that are not declared in COMMON. The Data Area is not preserved during chaining. The Data Area reserves two bytes for each integer and eight bytes for each real number. The Data Area stores the pointers that refer to strings and arrays in the FSA.
- The Computational Stack Area (CSA) is fixed at 100 bytes of memory. The CSA evaluates expressions and passes parameters to CBASIC predefined functions.
- The Free Storage Area (FSA) stores arrays, strings, and file buffers. Variably sized blocks of memory are allocated from the FSA as required and returned when no longer needed.

The starting and ending addresses for each partition in the TPA varies for different programs. Once allocated, however, the amount of memory each partition occupies remains fixed during program execution.

5.2 Internal Data Representation

CBASIC machine-level representation varies somewhat for real numbers, integers, strings, and arrays.

- REAL NUMBERS are stored in binary coded decimal (BCD) floating-point form. Each real number occupies eight bytes of memory storage space, as shown in Figure 5-2. The high-order bit in the first byte (byte 0) contains the sign of the number. The remaining seven bits in byte 0 contain a decimal exponent. Bytes 1 through 7 contain the mantissa. Two BCD digits occupy each of the seven bytes in the mantissa. The number's most significant digit is stored in byte 7, furthest from the exponent. The floating decimal point is always situated to the left of the most significant digit.

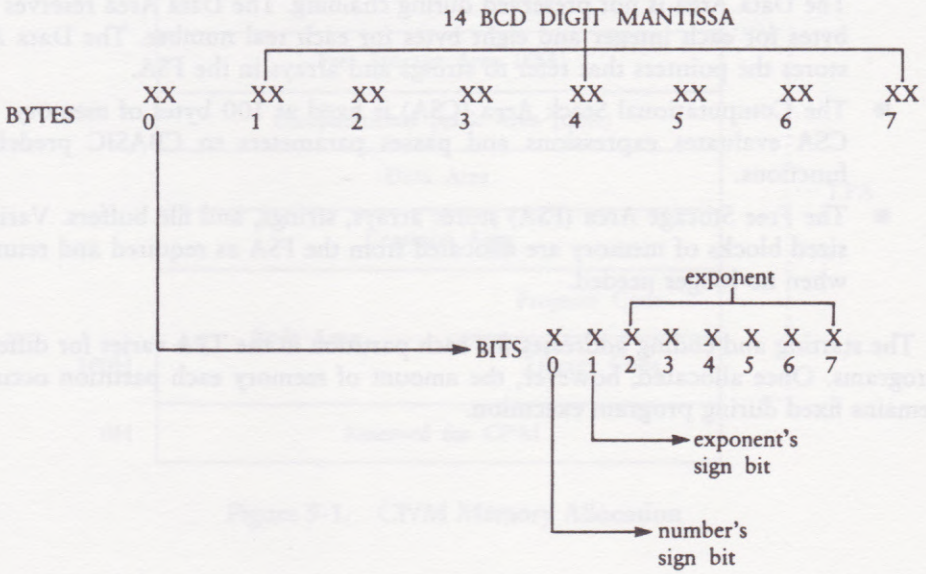

Figure 5-2. Real Number Storage

- INTEGERS are stored in two bytes of memory space with the low order byte first as shown in Figure 5-3. Integers are represented as 16-bit two's complement binary numbers. Integer values range from −32768 to +32767, inclusive.

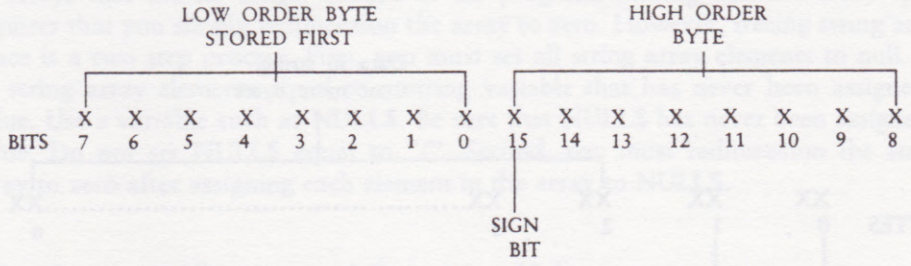

Figure 5-3. Integer Storage

- STRINGS are stored as a sequential list of ASCII representations. The length of a string is stored in the first two bytes followed by the actual ASCII values. The maximum number of characters in a string is 32,762. CBASIC Compiler allocates space in the Free Storage Area for strings. A pointer in the Data Area is an address in the FSA for the actual string.

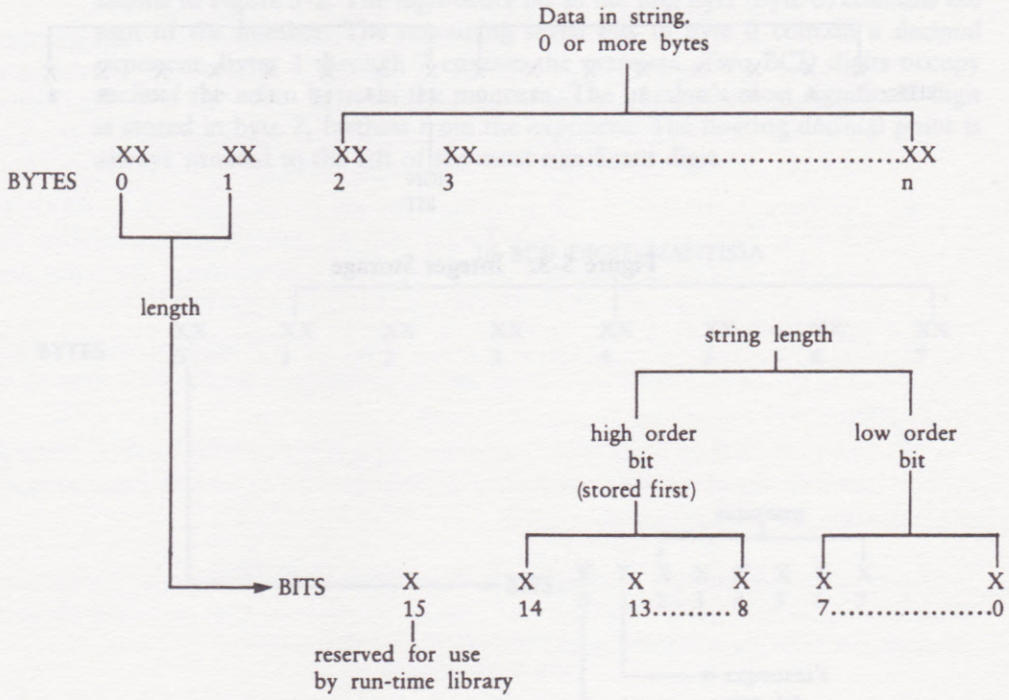

Figure 5-4. String Storage

Note: string lengths are stored high order and then low order. This is contrary to the normal 8080 convention for storing 16-bit quantities. The reserved bit, 15, is used to indicate that the string is temporary if the bit is a 1, and not temporary if it is a 0.

- ARRAYS, both numeric and string, are allocated space in the Free Storage Area as required. Eight bytes are reserved for each element of an array containing real numbers and two bytes for each element of an integer array.

At some point in a program, it might be necessary to free memory space allocated to arrays that are no longer needed in the program. Freeing numeric array space requires that you simply redimension the array to zero. However, freeing string array space is a two step process. First, you must set all string array elements to null. Set all string array elements equal to a string variable that has never been assigned a value. Use a variable such as NULL$. Be sure that NULL$ has never been assigned a value. Do not set NULL$ equal to "/". Second, you must redimension the string array to zero after assigning each element in the array to NULL$.

5.3 Parameter Passing and Returning Values

CBASIC Compiler passes all parameters on the hardware stack. When a program calls a routine, CBASIC places each parameter on the stack reading from left to right. The last entry on the stack is the return address. All values must conform to the format described in Section 5.2.

An assembly language routine can return integer, real, or string values to a CBASIC program. Before returning to the CBASIC program, all parameters passed on the stack must be removed and the stack pointer adjusted accordingly.

Integers return in registers H and L. Real numbers return using a pointer in registers H and L that points to an eight byte area containing the real value. The H and L registers contain the address of the exponent byte of the number being returned.

Strings return using a pointer in registers H and L. Strings must have been allocated using CBASIC Compiler dynamic storage management routines. The allocation bit of a returning string should be set to 1. This ensures that the space is initialized when no longer needed.

5.4 .REL File Format

CB80 and the Digital Research Relocating Macro Assembler (RMAC) create .REL files. A .REL file contains information encoded in a bit stream. You can interpret the information as described in the following list:

- If the first bit is a 0, the next 8 bits load according to the value of the location counter.
- If the first bit is a 1, the next two bits can be interpreted as follows:

  00 Special link item.

  01 Program relative. The next 16 bits load following an offset from the program segment origin.

  10 Data relative. The next 16 bits load following an offset from the data segment origin.

  11 Common relative. The next 16 bits load following an offset from the origin of the currently selected common block.

A special item consists of the following:

- 4-bit control field that selects one of 16 special link items described in Table 5-1.
- An optional value field that consists of a 2-bit address type field and a 16-bit address field. The address type field can be interpreted as follows:

  00 - absolute
  01 - program relative
  10 - data relative
  11 - common relative

- an optional name field consisting of a 3-bit name count followed by the name in 8-bit ASCII characters.

Table 5-1. Special Link Items

Control Field	*Meaning*
A name field follows the next five Link Items:	
0000	Entry symbol. This module describes the symbol indicated in the name field.
0001	Currently unassigned.
0010	Program name. This is the name of the relocatable module.
0011	Name field. Gives the name of default library file to use. LK80 assumes a filetype of .IRL.
0100	Currently unassigned.
A value field and a name field follow the next four link items:	
0101	Define common size. The value field determines the amount of memory to reserve for the common block indicated in the name field.
0110	Chain external. The value field contains the head of a chain that ends with an absolute 0. The value of the external symbol described in the name field replaces each element of the chain.
0111	Define entry point. The value field defines the value of the symbol in the name field. This link item puts local symbols in .REL files.
1000	Currently unassigned.

Table 5-1. (continued)

Control Field	*Meaning*
A value field follows the next six link items:	
1001	External plus offset. The value in the value field after all chains are processed must offset the following two bytes in the current segment.
1010	Define data size. The value field contains the number of bytes in the data segment of the current module.
1011	Set location counter. Set the location counter to the value indicated in the value field.
1100	Chain address. The value field contains the head of a chain that ends with an absolute 0. The current value of the location counter replaces each element of the chain.
1101	Define program size. The value field contains the number of bytes in the code segment of the current module.
1110	End module. Defines the end of the current module. If the value field contains a value other than absolute 0, the value is the start address for the linking program. The next item in the file will start at the next byte boundary.
The last item has no value field or name field:	
1111	End file. Follows the last module item for the last module in the file.

5.5 .IRL File Format

The CBASIC Compiler librarian utility, LIB.COM, creates .IRL files. An .IRL file consists of three parts: the header, the index, and the relocatable section. The header contains 128 bytes allocated as follows:

- byte 0 - extent number of first record of relocatable section
- byte 1 - record number of first record of relocatable section
- bytes 2 to 127 - currently unassigned

The index consists of entries that correspond to the entry symbol items listed in the relocatable section. Entries use the following form:

e	r	b	c1	c2	. . .	cn	d

e = Extent offset from start of relocatable section to start of module.

r = Record offset from start of extent to start of module.

b = Byte offset from start of record to start of module.

c1 - cn = Name of symbol.

d = End of symbol delimiter (0FEH).

When c1 equals 0FFH, the index terminates and the remainder of the record is not used.

The relocatable section contains relocatable object code as described in the Section 5.4.

End of Section 5

Appendix A
Implementation Dependent Values

The following implementation dependent values apply to CB80 version 1 for use with CP/M, versions 2 and 3, and MP/M-80™, versions 1 and 2:

Table A-1. Implementation Dependent Values

Parameter	*Value*	*Minimum*
Initial page width for compiler output	80	—
Initial page length for compiler output	66	—
Maximum number of errors maintained	95	—
Maximum nesting of include	6	4
Maximum number of formal parameters	15	15
Maximum number of subscripts in an array	15	15
Maximum unique identifier length	50	31
Maximum number of characters in string constant	255	255
Maximum length of Global and External names	6	6
Maximum nesting of FOR loops	13	—
Maximum nesting of WHILE loops	39	—
Number of files that can be open at one time	20	12
File buffer size in bytes	128	—

The minimum values are the minimum that are used in any CB80 implementation.

The following extensions exist in CB80, versions 1.3 and 1.4, to provide compatibility with CBASIC version 2. Note that future versions of CB80 might not support these extensions.

- The LPRINTER statement accepts a WIDTH option to be consistent with CBASIC. The width is ignored.
- Integer and real data is initialized to 0; strings are initialized to null strings. See Section 5.2.
- The INPUT prompt string can be any expression; the first operand must be a string constant.
- An OPEN or CREATE statement accepts a RECS field for compatibility with CBASIC. The expression is ignored.
- You can use the reserved words LT, GT, LE, GE, EQ, and NE in place of the relational operators <, >, <=, >=, =, and <>.
- CB80 supports the following form of an IF statement:

  IF *expression* THEN *label*

  but the *label* must be a numeric label.

End of Appendix A

Appendix B
Compiler Error Messages

The compiler prints the following messages when a file system error or memory space error occurs. In each case, control returns to the operating system.

Table B-1. File System and Memory Space Errors

Error	*Meaning*
COULD NOT OPEN FILE:*filename*	The filename following the message cannot be located in the file system directory.
%INCLUDES NESTED TOO DEEP:*filename*	The filename following the message occurred in an %INCLUDE directive that exceeds the allowed nesting of %INCLUDE directives.
SYMBOL TABLE OVERFLOW	The available memory for symbol table space has been exceeded. Break the program into modules or use shorter symbol names.
INVALID FILE NAME:*filename*	The filename is not valid for your operating system.
DISK READ ERROR	The operating system reports a disk read error.

Table B-1. (continued)

Error	*Meaning*
`CREATE ERROR:` *filename*	The file cannot be created. Normally this means there is no directory space on the disk.
`DISK FULL`	The operating system reports that no additional space is available to write temporary or output files. The directory may be full or the disk is out of space.
`INVALID COMMAND LINE`	The command line is incorrect. The compiler prints a greater-than sign, >, one blank space, and all command line characters beginning with the first character in error. If no characters remain in the command line when an error occurs, the compiler does not print the > or the space.
`MISSING SOURCE FILE NAME`	The command line processor reports that you did not specify a source file.
`CLOSE OR DELETE ERROR`	The operating system reports that it cannot close a file. This occurs if diskettes are switched during compilation.

If the compiler detects an internal failure, the following error message appears:

```
FATAL COMPILER ERROR XXX
NEAR SOURCE LINE XXXX
```

where XXX is a three digit number. Please advise Digital Research of the error and the circumstances under which it occurs.

The following error messages indicate a compilation error occurred during compilation of a program. Compilation continues after the error is recorded. Compilation error messages display within the source code listing.

Table B-2. Compilation Error Messages

Error	*Meaning*
1	Invalid character in the source program. The character is ignored.
2	Invalid string constant. The string is too long or contains a carriage return.
3	Invalid numeric constant. An integer constant of zero is assumed.
4	Undefined compiler directive. This source line is ignored.
5	The %INCLUDE directive is missing a filename. This source line is ignored.
6	Statements found after an END.
7	Not used.
8	Variable used without being defined, and the U toggle used during compilation.
9	The DEF statement is not terminated by a carriage return. A carriage return is inserted.
10	A right parenthesis is missing from the parameter list. A right parenthesis is inserted.
11	A comma is missing in the parameter list. A comma is inserted.
12	An identifier is missing in the parameter list.
13	The same name is used twice in a parameter list.
14	A DEF statement occurs within a multiple-line function. Multiple-line functions cannot be nested. The statement is ignored.

Table B-2. (continued)

Error	*Meaning*
15	A variable is missing.
16	The function name is missing following the keyword DEF. The DEF statement is ignored.
17	A function name is used previously. The DEF statement is ignored.
18	A FEND statement is missing. A FEND is inserted.
19	There are too many parameters in a multiple line function.
20	Inconsistent identifier usage. An identifier cannot be used as both a label and a variable.
21	Additional data exists in the source file following an END statement. This is the logical end of the program.
22	Data statements must begin on a new line. The remainder of this statement is treated as a remark.
23	There are no variables or function names in a declaration statement; or, a reserved word appears in the list of identifiers.
24	A function name appears in a declaration within a multiple-line function other than the multiple-line function that defines this function name.
25	A function call has incorrect number of parameters.
26	A left parenthesis is missing. A left parenthesis is inserted.
27	Invalid mixed mode. The type of the expression is not permitted.
28	Unary operator cannot be used with this operand.
29	Function call has improper type of parameter.

Table B-2. (continued)

Error	*Meaning*
30	Invalid symbol follows a variable, constant, or function reference.
31	This symbol cannot occur at this location in an expression. The symbol is ignored.
32	Operator is missing. Multiplication operator inserted.
33	Invalid symbol encountered in an expression. The symbol is ignored.
34	A right parenthesis is missing. A right parenthesis is inserted.
35	A subscripted variable is used with the incorrect number of subscripts.
36	An identifier is used as a simple variable with previous usage as a subscripted variable.
37	An identifier is used as a subscripted variable with previous usage as an unsubscripted variable.
38	A string expression is used as a subscript in an array reference.
39	A constant is missing.
40	Invalid symbol found in declaration list. The symbol is skipped.
41	A carriage return is missing in a declaration statement. A carriage return is inserted.
42	Comma is missing in declaration list. A comma is inserted.
43	A common declaration cannot occur in a multiple-line function. The statement is ignored.
44	An identifier appears in a declaration twice in the main program or within the same multiple-line function.

Table B-2. (continued)

Error	*Meaning*
45	The number of dimensions specified for an array exceeds the maximum number allowed. A value of one is used. This might generate additional errors in the program.
46	Right parenthesis is missing in the dimension specification within a declaration. A right parenthesis is inserted.
47	The same identifier is placed in COMMON twice.
48	An invalid subscripted variable reference encountered in a declaration statement. An integer constant is required. A value of 1 is used.
49	An invalid symbol found following a declaration, or the symbol in the first statement in the program is invalid. The symbol is ignored.
50	An invalid symbol encountered at the beginning of a statement or following a label.
51	An equal sign is missing in assignment. An equal sign is inserted.
52	A name used as a label previously used at this level as either a label or variable.
53	Unexpected symbol follows a simple statement. The symbol is ignored.
54	A statement is not terminated with a carriage return. Text is ignored until the next carriage return.
55	A function name is used in the left part of an assignment statement outside of a multiple-line function. Only when the function is being compiled can its name appear on the left of an assignment statement.
56	A predefined function name is used as the left part of an assignment statement.
57	In an IF statement, a THEN is missing. A THEN is inserted.

Table B-2. (continued)

Error	*Meaning*
58	A WEND statement is missing. A WEND is inserted.
59	A carriage return or colon is missing at the end of a WHILE loop header.
60	In a FOR loop header the index is missing. The compiler skips to end of this statement.
61	In a FOR loop header, a TO is missing. A TO is inserted.
62	An equal sign is missing in a FOR loop header assignment. An equal sign is inserted.
63	Carriage return or colon is missing at end of FOR loop header.
64	A NEXT statement is missing. A NEXT is inserted.
65	Not used.
66	The variable that follows NEXT does not match the FOR loop index.
67	NEXT statement encountered without a corresponding FOR loop header.
68	WEND statement encountered without a corresponding WHILE loop header.
69	FEND statement encountered without a corresponding DEF statement. This error indicates that the end of the source program was detected while within a multiple-line function.
70	The PRINT USING string is not of type string.
71	A delimiter is missing in a PRINT statement. A comma is inserted.
72	A semicolon is missing in an INPUT prompt. A semicolon is inserted.

Table B-2. (continued)

Error	*Meaning*
73	A delimiter is missing in an INPUT statement. A comma is inserted.
74	A semicolon is missing following a file reference. A semicolon is inserted.
75	The prompt in an INPUT statement is not of type string.
76	In an INPUT LINE statement, the variable following the keyword LINE is not a string variable.
77	In an INPUT statement, a comma is missing between variables. A comma is inserted.
78	The keyword AS is missing in an OPEN or CREATE statement. AS is inserted.
79	The filename in an OPEN or CREATE statement is not a string expression.
80	A delimiter is missing in a READ statement. A comma is inserted.
81	In a GOTO, GOSUB, or ON statement, a label is missing. This token can be an identifier previously used as a variable.
82	The label in a GOTO statement is not defined. If the label is used in a function, it must be defined in that function.
83	A delimiter is missing in a file READ statement. A comma is inserted.
84	In a READ LINE statement, the variable following the keyword LINE is not a string variable.
85	The label in an IF END statement is not defined.
86	A pound sign, #, is missing in an IF END statement. A pound sign is inserted.

Table B-2. (continued)

Error	*Meaning*
87	A THEN is missing in an IF END statement. A THEN is inserted.
88	In a PRINT statement, the semicolon is missing following a using string. A semicolon is inserted.
89	In an ON statement, a GOTO or GOSUB is missing. A GOTO is assumed.
90	The index of a FOR loop header is of type string. The index must be an integer or real value.
91	The expression following the keyword TO in a FOR loop header is of type string. The expression must be an integer or real value.
92	The expression following the keyword STEP in a FOR loop header is of type string. The expression must be an integer or real value.
93	A variable in a DIM statement is defined previously as other than a subscripted variable.
94	An identifier is missing as an array name in a DIM statement. The entire statement is ignored.
95	A left parenthesis is missing in a DIM statement. A left parenthesis is inserted.
96	A right parenthesis is missing in a DIM statement. A right parenthesis is inserted.
97	The maximum number of dimensions allowed with a subscripted variable is exceeded.
98	A comma is missing in a POKE statement. A comma is inserted.
99	The index of a FOR loop header is not a simple variable.
100	In a CALL statement, a multiple-line function name is missing.

Table B-2. (continued)

Error	*Meaning*
101	A file PRINT statement is terminated with a comma or semicolon.
102	A DIM statement is missing for this subscripted variable.
103	A comma is missing in the label list associated with an ON GOTO or ON GOSUB statement. A comma is inserted.
104	A GOTO is missing in an ON ERROR statement. A GOTO is inserted.
105	A comma is missing in a PUT statement. A comma is inserted.
106	The expression in an IF statement is of type string. An integer or real expression is required.
107	The expression in a WHILE loop header is of type string. An integer or real expression is required.
108	In an OPEN or CREATE statement, the filename is missing.
109	In an OPEN or CREATE statement, the expression following the reserved word AS is missing.

Table B-2. (continued)

Error	*Meaning*
110	A multiple-line function calls itself.
111	A semicolon separates expressions in a file PRINT statement. A comma is substituted for the semicolon.
112	A file PRINT statement does not have an expression list.
113	A TAB function is used in a file PRINT statement expression list.
114	Label used as a variable in a list of expressions.
115	A GO not followed by a TO or SUB. GOTO is assumed.
116	An OPEN or CREATE statement specifies both UNLOCKED and LOCKED access control.
117	A CREATE statement uses the READ ONLY access control.

End of Appendix B

Appendix C
LK80 Error Messages

LK80 prints the following messages to indicate the occurrence of an error during linking. Control returns to the operating system after the message is displayed.

Table C-1. LK80 Error Messages

Message	*Meaning*
`Unresolved external:` *symbol name*	You defined the symbol name as external but neglected to define the symbol as public.
`Out of Directory Space`	LK80 ran out of directory space while writing the root, overlay, symbol, or environment file.
`Disk Full`	LK80 ran out of disk space while writing the root, overlay, symbol, or environment file.
`Multiple Definition:` *symbol name*	You defined a symbol name more than once.
`Too many overlays`	You specified more than 60 overlays in the command line.
`Too many modules`	You specified more than 60 modules in the command line.

Table C-1. (continued)

Message	*Meaning*
`Symbol table overflow`	There is not enough memory for the symbol table.
`Cannot open source file`	A source file specified in the command line cannot be opened.
`Too many library files`	You cannot specify more than 10 indexed library files in the command line.
`Too many library modules`	You cannot extract more than 150 modules from all indexed library files.
`Index too big`	A library file index cannot exceed 16K.
`Too many external-plus-offsets`	The table that saves external-plus-offsets has overflowed. References to offsets from external symbols usually occur in assembly language programs.
`Code size exceeded. Short link aborted.`	The new overlay cannot require a code segment larger than the code segment in the original full link.
`Data size exceeded. Short link aborted.`	The new overlay cannot require a data segment larger than the data segment in the original full link.

Table C-1. (continued)

Message	*Meaning*
`Common size exceeded. Short link aborted.`	The new overlay cannot require a common segment larger than the common segment in the original full link.
`Root has no entry point.`	You did not specify the root program in the command line or your root program does not contain executable statements.
`No entry point defined for overlay:` *overlay*	The overlay file specified in the message does not contain executable statements.
`Not enough memory`	There is not enough memory for LK80 to complete linking the modules specified in the command line.
`Cannot close file:` *file*	LK80 cannot complete linking because it cannot close the module specified in the message.
`Expected module name`	You did not specify a module name in the command line.
`Toggle not supported`	You specified an invalid toggle letter in the command line.
`Expected ] at end of toggle definition`	You omitted a closing square bracket in a command line toggle definition.

Table C-1. (continued)

Message	*Meaning*
`Unexpected ( ?`	You entered a left parenthesis without a matching right parenthesis in the command line, indicating an incomplete overlay specification.
`Unexpected ) ?`	You entered a right parenthesis without a matching left parenthesis in the command line, indicating an incomplete overlay specification.
`Invalid or unexpected character`	You entered a character in the command line that LK80 does not recognize or did not expect at a certain position in the command line.
`Module name or type too long`	Module names cannot exceed 8 characters and types cannot exceed 3 characters.
`Can only specify output name on first module`	Only one module name can precede the equal sign in a command line. If you do not use the equal sign, the first module listed becomes the name of the output file.
`Multiple entry points in:` *filespec*	More than one file specified in the command line contains executable statements. The file specified in the message contains executable statements.

End of Appendix C

Appendix D
Execution Error Messages

The following warning message might be printed during execution of a CB80 program:

```
IMPROPER INPUT - REENTER
```

This message occurs when the fields you enter from the console do not match the fields specified in the INPUT statement. Following this message, you must reenter all values required by the input statement.

Execution errors cause a two-letter code to be printed. The following table contains valid CB80 error codes.

If an error occurs with a code consisting of an asterisk followed by a letter, such as *R, a CB80 library has failed. Please notify Digital Research of the circumstances under which the error occurred.

Table D-1. CB80 Error Codes

Code	*Error*
AC	The argument in an ASC function is a null string.
BN	The value following the BUFF option in an OPEN or CREATE statement is less than 1 or greater than 128.
CE	The file being closed cannot be found in the directory. This occurs if the file has been changed by the RENAME function.
CM	The file specified in a CHAIN statement cannot be found in the selected directory. If no filetype is present, the compiler assumes a type of OVL.
CT	The filetype of the file specified in a CHAIN statement is other than COM or OVL.

Table D-1. (continued)

Code	*Error*
CU	A CLOSE statement specifies a file identification number that is not active.
DF	An OPEN or CREATE statement uses a file identification number that is already used.
DU	A DELETE statement specifies a file identification number that is not active.
DW	The operating system reports that there is no disk or directory space available for the file being written to, and no IF END statement is in effect for the file identification number.
DZ	Division by zero is attempted.
EF	Attempt to read past the end-of-file, and no IF END statement is in effect for the file identification number.
ER	Attempt to write a record of length greater than the maximum record size specified in the OPEN or CREATE statement for this file.
EX	Indicates MP/M II extended error.
FR	Attempt to rename a file to a filename that already exists.
FU	Attempt to access a file that was not open.
IF	A filename in an OPEN or CREATE statement or with the RENAME function is invalid for your operating system.
IR	A record number of zero is specified in a READ or PRINT statement.
LN	The argument in the LOG function is zero or negative.
ME	The operating system reports an error during an attempt to create or extend a file. Normally, this means the disk directory is full.

Table D-1. (continued)

Code	*Error*
MP	The third parameter in a MATCH function is zero or negative.
NE	A negative value is specified for the operand to the left of the power operator.
NF	A file identification is less than 1 or greater than the maximum number allowed. See Appendix E.
NN	An attempt to print a numeric expression with a PRINT USING statement fails because there is not a numeric field in the USING string.
NS	An attempt to print a string expression with a PRINT USING statement fails because there is not a string field in the USING string.
OD	A READ statement is executed but there are no DATA statements in the program, or all data items in all the DATA statements have been read.
OE	Attempt to OPEN a file that does not exist, and for which no IF END statement is in effect.
OF	An overflow occurs during a real arithmetic calculation.
OM	The program runs out of dynamically allocated memory during execution.
RB	Random access is attempted to a file activated with the BUFF option specifying more than one buffer.
RE	Attempt to read past the end of a record in a fixed file.
RU	A random read or print is attempted to a stream file.

Table D-1. (continued)

Code	*Error*
SL	A concatenation operation results in a string greater than the maximum allowed string length.
SQ	Attempt to calculate the square root of a negative number.
SS	The second parameter of a MID$ function is zero or negative, or the last parameter of a LEFT$, RIGHT$, or MID$ is negative.
TL	A tab statement contains a parameter less than 1.
UN	A PRINT USING statement is executed with a null edit string, or the backslash escape character, \, is the last character in an edit string.
WR	Attempt to write to a stream file after it is read, but before it is read to the end-of-file.

End of Appendix D

Appendix E
LIB Error Messages

The following table presents the LIB.COM program error messages and descriptions.

Table E-1. LIB Error Messages

Message	*Meaning*
CANNOT CLOSE	LIB cannot close the output file. The diskette might be write-protected.
DIRECTORY FULL	There is no directory space for the output file.
DISK READ ERROR	LIB cannot read the specified file.
DISK WRITE ERROR	LIB cannot write the specified file; probably due to a full diskette.
FILE NAME ERROR	The form of a source filename is invalid.
NO FILE	LIB cannot find the file that is specified in the command line.
NO MODULE	LIB cannot find the module that is specified in the command line.
SYNTAX ERROR	You used an incorrect command line to start the LIB program.

End of Appendix E

Index

U

V

W

X